On partial order semantics for SAT/SMT-based symbolic encodings of weak memory concurrency*

Alex Horn and Daniel Kroening

University of Oxford

Abstract. Concurrent systems are notoriously difficult to analyze, and technological advances such as weak memory architectures greatly compound this problem. This has renewed interest in partial order semantics as a theoretical foundation for formal verification techniques. Among these, symbolic techniques have been shown to be particularly effective at finding concurrency-related bugs because they can leverage highly optimized decision procedures such as SAT/SMT solvers. This paper gives new fundamental results on partial order semantics for SAT/SMT-based symbolic encodings of weak memory concurrency. In particular, we give the theoretical basis for a decision procedure that can handle a fragment of concurrent programs endowed with least fixed point operators. In addition, we show that a certain partial order semantics of relaxed sequential consistency is equivalent to the conjunction of three extensively studied weak memory axioms by Alglave et al. An important consequence of this equivalence is an asymptotically smaller symbolic encoding for bounded model checking which has only a quadratic number of partial order constraints compared to the state-of-the-art cubic-size encoding.

1 Introduction

Concurrent systems are notoriously difficult to analyze, and technological advances such as weak memory architectures as well as highly available distributed services greatly compound this problem. This has renewed interest in partial order concurrency semantics as a theoretical foundation for formal verification techniques. Among these, *symbolic techniques* have been shown to be particularly effective at finding concurrency-related bugs because they can leverage highly optimized decision procedures such as SAT/SMT solvers. This paper studies partial order semantics from the perspective of SAT/SMT-based symbolic encodings of weak memory concurrency.

Given the diverse range of partial order concurrency semantics, we link our study to a recently developed unifying theory of concurrency by Tony Hoare et al. [1]. This theory is known as *Concurrent Kleene Algebra* (CKA) which is an algebraic concurrency semantics based on quantales, a special case of the fundamental algebraic structure of idempotent semirings. Based on quantales, CKA combines the familiar laws of the sequential program operator (;) with a

* This work is funded by a gift from Intel Corporation for research on Effective Validation of Firmware and the ERC project ERC 280053.

new operator for concurrent program composition ($\|$). A distinguishing feature of CKA is its exchange law $(\mathcal{U} \parallel \mathcal{V}); (\mathcal{X} \parallel \mathcal{Y}) \subseteq (\mathcal{U}; \mathcal{X}) \parallel (\mathcal{V}; \mathcal{Y})$ that describes how sequential and concurrent composition operators can be interchanged. Intuitively, since the binary relation $\subseteq$ denotes program refinement, the exchange law expresses a divide-and-conquer mechanism for how concurrency may be sequentially implemented on a machine. The exchange law, together with a uniform treatment of programs and their specifications, is key to unifying existing theories of concurrency [2]. CKA provides such a unifying theory [3,2] that has practical relevance on proving program correctness, e.g. using rely/guarantee reasoning [1]. Conversely, however, pure algebra cannot refute that a program is correct or that certain properties about every program always hold [3,2,4]. This is problematic for theoretical reasons but also in practice because todays software complexity requires a diverse set of program analysis tools that range from proof assistants to automated testing. The solution is to accompany CKA with a mathematical model which satisfies its laws so that we can *prove* as well as *disprove* properties about programs.

One such well-known model-theoretical foundation for CKA is Pratt's [5] and Gischer's [6] partial order model of computation that is constructed from *labelled partially ordered multisets* (pomsets). Pomsets generalize the concept of a string in finite automata theory by relaxing the total ordering of the occurrence of letters within a string to a partial order. For example, $a \parallel a$ denotes a pomset that consists of two unordered events that are both labelled with the letter a. By partially ordering events, pomsets form an integral part of the extensive theoretical literature on so-called 'true concurrency', e.g. [7,8,9,10,5,6], in which pomsets strictly generalize Mazurkiewicz traces [11], and prime event structures [10] are pomsets enriched with a conflict relation subject to certain conditions. From an algorithmic point of view, the complexity of the *pomset language membership* (PLM) problem is NP-complete, whereas the pomset language containment (PLC) problem is Π_2^p-complete [12].

Importantly, these aforementioned theoretical results only apply to star-free pomset languages (without fixed point operators). In fact, the decidability of the equational theory of the pomset language closed under least fixed point, sequential and concurrent composition operators (but without the exchange law) has been only most recently established [13]; its complexity remains an open problem [13]. Yet another open problem is the decidability of this equational theory together with the exchange law [13]. In addition, it is still unclear how theoretical results about pomsets may be applicable to formal techniques for finding concurrency-related bugs. In fact, it is not even clear how insights about pomsets may be combined with most recently studied language-specific or hardware-specific concurrency semantics, e.g. [14,15,16,17].

These gaps are motivation to reinvestigate pomsets from an algorithmic perspective. In particular, our work connects pomsets to a SAT/SMT-based bounded model checking technique [18] where shared memory concurrency is symbolically encoded as partial orders. To make this connection, we adopt pomsets as *partial strings* (Definition 1) that are ordered by a refinement relation (Definition 3) based on Ésik's notion of *monotonic bijective morphisms* [19]. Our partial-string model then follows from the standard Hoare powerdomain construction where sets of partial strings are downward-closed with respect to monotonic bijective morphism (Definition 4). The relevance of this formalization for the modelling of weak memory concurrency (including data races) is explained through several examples. Our main contributions are as follows:

1. We give the theoretical basis for a decision procedure that can handle a fragment of *concurrent programs endowed with least fixed point operators* (Theorem 2). This is accomplished by exploiting a form of periodicity, thereby giving a mechanism for reducing a countably infinite number of events to a finite number. This result particularly caters to partial order encoding techniques that can currently only encode a finite number of events due to the deliberate restriction to quantifier-free first-order logic, e.g. [18].
2. We then interpret a particular form of weak memory in terms of certain downward-closed sets of partial strings (Definition 11), and show that our interpretation is equivalent to the conjunction of three fundamental weak memory axioms (Theorem 3), namely 'write coherence', 'from-read' and 'global read-from' [17]. Since all three axioms underpin extensive experimental research into weak memory architectures [20], *Theorem 3 gives denotational partial order semantics a new practical dimension*.
3. Finally, we prove that there exists an *asymptotically smaller quantifier-free first-order logic formula* that has only $O(N^2)$ partial order constraints (Theorem 4) compared to the state-of-the-art $O(N^3)$ partial order encoding for bounded model checking [18] where N is the maximal number of reads and writes on the same shared memory address. This is significant because N can be prohibitively large when concurrent programs frequently share data.

The rest of this paper is organized into three parts. First, we recall familiar concepts on partial-string theory (§ 2) on which the rest of this paper is based. We then prove a least fixed point reduction result (§ 3). Finally, we characterize a particular form of relaxed sequential consistency in terms of three weak memory axioms by Alglave et al. (§ 4).

2 Partial-string theory

In this section, we adapt an axiomatic model of computation that uses partial orders to describe the semantics of concurrent systems. For this, we recall familiar concepts (Definition 1, 2, 3 and 4) that underpin our mathematical model of CKA (Theorem 1). This model is the basis for subsequent results in § 3 and § 4.

Definition 1 (Partial string). *Denote with E a nonempty set of **events**. Let Γ be an **alphabet**. A **partial string** p is a triple $\langle E_p, \alpha_p, \preceq_p \rangle$ where E_p is a subset of E, $\alpha_p \colon E_p \to \Gamma$ is a function that maps each event in E_p to an alphabet symbol in Γ, and $\preceq_p$ is a partial order on E_p. Two partial strings p and q are said to be **disjoint** whenever $E_p \cap E_q = \emptyset$. A partial string p is called **empty** whenever $E_p = \emptyset$. Denote with P_f the set of all **finite partial strings** p whose event set E_p is finite.*

$$\begin{tikzcd} e_0 \arrow[d] & e_2 \arrow[d] \\ e_1 & e_3 \end{tikzcd}$$

Fig. 1. A partial string $p = \langle E_p, \alpha_p, \preceq_p \rangle$ with events $E_p = \{e_0, e_1, e_2, e_3\}$ and the labelling function α_p satisfying the following: $\alpha_p(e_0) =$ '$r_0 := [b]_{\mathsf{acquire}}$', $\alpha_p(e_1) =$ '$r_1 := [a]_{\mathsf{none}}$', $\alpha_p(e_2) =$ '$[a]_{\mathsf{none}} := 1$' and $\alpha_p(e_3) =$ '$[b]_{\mathsf{release}} := 1$'.

Each event in the universe E should be thought of as an occurrence of a computational step, whereas letters in Γ describe the computational effect of events. Typically, we denote a partial string by p, or letters from x through z. In essence, a partial string p is a partially-ordered set $\langle E_p, \preceq_p \rangle$ equipped with a labelling function α_p. A partial string is therefore the same as a *labelled partial order* (lpo), see also Remark 1. We draw finite partial strings in P_f as inverted Hasse diagrams (e.g. Fig. 1), where the ordering between events may be interpreted as a happens-before relation [8], a fundamental notion in distributed systems and formal verification of concurrent systems, e.g. [16,17]. We remark the obvious fact that the empty partial string is unique under component-wise equality.

Example 1. In the partial string in Fig. 1, e_0 happens-before e_1, whereas both e_0 and e_2 happen concurrently because neither $e_0 \preceq_p e_2$ nor $e_2 \preceq_p e_0$.

We abstractly describe the control flow in concurrent systems by adopting the sequential and concurrent operators on labelled partial orders [9,5,6,19,21].

Definition 2 (Partial string operators). *Let x and y be disjoint partial strings. Let $x \parallel y \triangleq \langle E_{x\parallel y}, \alpha_{x\parallel y}, \preceq_{x\parallel y} \rangle$ and $x ; y \triangleq \langle E_{x;y}, \alpha_{x;y}, \preceq_{x;y} \rangle$ be their **concurrent** and **sequential composition**, respectively, where $E_{x\parallel y} = E_{x;y} \triangleq E_x \cup E_y$ such that, for all events e, e' in $E_x \cup E_y$, the following holds:*

- *$e \preceq_{x\parallel y} e'$ exactly if $e \preceq_x e'$ or $e \preceq_y e'$,*
- *$e \preceq_{x;y} e'$ exactly if $(e \in E_x$ and $e' \in E_y)$ or $e \preceq_{x\parallel y} e'$,*
- $\alpha_{x\parallel y}(e) = \alpha_{x;y}(e) \triangleq \begin{cases} \alpha_x(e) & \text{if } e \in E_x \\ \alpha_y(e) & \text{if } e \in E_y. \end{cases}$

For simplicity, we assume that partial strings can be always made disjoint by renaming events if necessary. But this assumption could be avoided by using coproducts, a form of constructive disjoint union [21]. When clear from the context, we construct partial strings directly from the labels in Γ.

Example 2. If we ignore labels for now and let p_i for all $0 \leq i \leq 3$ be four partial strings which each consist of a single event e_i, then $(p_0; p_1) \parallel (p_2; p_3)$ corresponds to a partial string that is isomorphic to the one shown in Fig. 1.

To formalize the set of all possible happens-before relations of a concurrent system, we rely on Ésik's notion of monotonic bijective morphism [19]:

Definition 3 (Partial string refinement). *Let x and y be partial strings such that $x = \langle E_x, \alpha_x \preceq_x \rangle$ and $y = \langle E_y, \alpha_y, \preceq_y \rangle$. A **monotonic bijective morphism** from x to y, written $f \colon x \to y$, is a bijective function f from E_x to E_y such that, for all events $e, e' \in E_x$, $\alpha_x(e) = \alpha_y(f(e))$, and if $e \preceq_x e'$, then $f(e) \preceq_y f(e')$. Then x **refines** y, written $x \sqsubseteq y$, if there exists a monotonic bijective morphism $f \colon y \to x$ from y to x.*

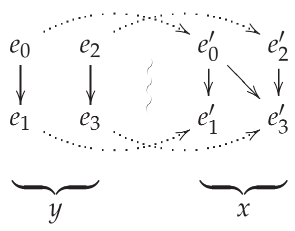

Fig. 2. Two partial strings x and y such that $x \sqsubseteq y$ provided all the labels are preserved, e.g. $\alpha_x(e'_0) = \alpha_y(e_0)$.

Remark 1. Partial words [9] and pomsets [5,6] are defined in terms of isomorphism classes of lpos. Unlike lpos in pomsets, however, we study partial strings in terms of monotonic bijective morphisms [19] because isomorphisms are about sameness whereas the exchange law on partial strings is an inequation [21].

The purpose of Definition 3 is to disregard the identity of events but retain the notion of 'subsumption', cf. [6]. The intuition is that $\sqsubseteq$ orders partial strings according to their determinism. In other words, $x \sqsubseteq y$ for partial strings x and y implies that all events ordered in y have the same order in x.

Example 3. Fig. 2 shows a monotonic bijective morphism from a partial string as given in Fig. 1 to an N-shaped partial string that is almost identical to the one in Fig. 1 except that it has an additional partial order constraint, giving its N shape. One well-known fact about N-shaped partial strings is that they cannot be constructed as $x;y$ or $x \parallel y$ under any labelling [5]. However, this is not a problem for our study, as will become clear after Definition 4.

Our notion of partial string refinement is particularly appealing for symbolic techniques of concurrency because the monotonic bijective morphism can be directly encoded as a first-order logic formula modulo the theory of uninterpreted functions. Such a symbolic partial order encoding would be fully justified from a computational complexity perspective, as shown next.

Proposition 1. *Let x and y be finite partial strings in P_f. The* ***partial string refinement*** *(PSR) problem — i.e. whether $x \sqsubseteq y$ — is NP-complete.*

Proof. Clearly PSR is in NP. The NP-hardness proof proceeds by reduction from the PLM problem [12]. Let Γ^* be the set of strings, i.e. the set of finite partial strings s such that $\preceq_s$ is a total order (for all $e, e' \in E_s$, $e \preceq_s e'$ or $e' \preceq_s e$). Given a finite partial string p, let $\mathfrak{L}_p$ be the set of all strings which refine p; equivalently, $\mathfrak{L}_p \triangleq \{s \in \Gamma^* \mid s \sqsubseteq p\}$. So $\mathfrak{L}_p$ denotes the same as $L(p)$ in [12, Definition 2.2].

Let s be a string in Γ^* and P be a pomset over the alphabet Γ. By Remark 1, fix p to be a partial string in P. Thus s refines p if and only if s is a member of $\mathfrak{L}_p$. Since this membership problem is NP-hard [12, Theorem 4.1], it follows that the PSR problem is NP-hard. So the PSR problem is NP-complete. $\square$

Note that a single partial string is not enough to model mutually exclusive (nondeterministic) control flow. To see this, consider a simple (possibly sequential) system such as `if * then P else Q` where `*` denotes nondeterministic choice. If the semantics of a program was a single partial string, then we need to find exactly one partial string that represents the fact that `P` executes or `Q` executes, but never both. To model this, rather than using a conflict relation [10], we resort to the simpler Hoare powerdomain construction where we lift sequential and concurrent composition operators to *sets* of partial strings. But since we are aiming (similar to Gischer [6]) at an *over-approximation of concurrent systems*, these sets are downward closed with respect to our partial string refinement ordering from Definition 3. Additional benefits of using the downward closure include that program refinement then coincides with familiar set inclusion and the ease with which later the Kleene star operators can be defined.

Definition 4 (Program). *A **program** is a downward-closed set of finite partial strings with respect to $\sqsubseteq$; equivalently $\mathcal{X} \subseteq \mathsf{P}_f$ is a program whenever $\downarrow_{\sqsubseteq} \mathcal{X} = \mathcal{X}$ where $\downarrow_{\sqsubseteq} \mathcal{X} \triangleq \{y \in \mathsf{P}_f \mid \exists x \in \mathcal{X} \colon y \sqsubseteq x\}$. Denote with $\mathbb{P}$ the family of all programs.*

Since we only consider systems that terminate, each partial string x in a program $\mathcal{X}$ is finite. We reemphasize that the downward closure of such a set $\mathcal{X}$ can be thought of as an over-approximation of all possible happens-before relations in a concurrent system whose instructions are ordered according to the partial strings in $\mathcal{X}$. Later on (§ 4) we make the downward closure of partial strings more precise to model a certain kind of relaxed sequential consistency.

Example 4. Recall that N-shaped partial strings cannot be constructed as $x\,;y$ or $x \parallel y$ under any labelling [5]. Yet, by downward-closure of programs, such partial strings are included in the over-approximation of all the happens-before relations exhibited by a concurrent system. In particular, according to Example 3, the downward-closure of the set containing the partial string in Fig. 1 includes (among many others) the N-shaped partial string shown on the right in Fig. 2. In fact, we shall see in § 4 that this particular N-shaped partial string corresponds to a data race in the concurrent system shown in Fig. 3.

It is standard [6,21] to define $0 \triangleq \emptyset$ and $1 \triangleq \{\bot\}$ where $\bot$ is the (unique) empty partial string. Clearly 0 and 1 form programs in the sense of Definition 4. For the next theorem, we lift the two partial string operators (Definition 2) to programs in the standard way:

Definition 5 (Bow tie). *Given two partial strings x and y, denote with $x \bowtie y$ either concurrent or sequential composition of x and y. For all programs $\mathcal{X}, \mathcal{Y}$ in $\mathbb{P}$ and partial string operators $\bowtie$, $\mathcal{X} \bowtie \mathcal{Y} \triangleq \downarrow_{\sqsubseteq} \{x \bowtie y \mid x \in \mathcal{X} \text{ and } y \in \mathcal{Y}\}$ where $\mathcal{X} \parallel \mathcal{Y}$ and $\mathcal{X}\,;\mathcal{Y}$ are called **concurrent** and **sequential program composition**, respectively.*

By denoting programs as sets of partial strings, we can now define Kleene star operators $(-)^{\parallel}$ and $(-)^{;}$ for iterative concurrent and sequential program composition, respectively, as least fixed points (μ) using set union ($\cup$) as the binary join operator that we interpret as the nondeterministic choice of two programs. We remark that this is fundamentally different from the pomsets recursion operators in ultra-metric spaces [22]. The next theorem could be then summarized as saying that the resulting structure of programs, written $\mathfrak{S}$, is a partial order model of an algebraic concurrency semantics that satisfies the CKA laws [1]. Since CKA is an exemplar of the universal laws of programming [2], we base the rest of this paper on our partial order model of CKA.

Theorem 1. *The structure $\mathfrak{S} = \langle \mathbb{P}, \subseteq, \cup, 0, 1, ;, \parallel \rangle$ is a complete lattice, ordered by subset inclusion (i.e. $\mathcal{X} \subseteq \mathcal{Y}$ exactly if $\mathcal{X} \cup \mathcal{Y} = \mathcal{Y}$), such that $\parallel$ and ; form unital quantales over $\cup$ where $\mathfrak{S}$ satisfies the following:*

$$\begin{array}{lr}
(\mathcal{U} \parallel \mathcal{V}); (\mathcal{X} \parallel \mathcal{Y}) \subseteq (\mathcal{U}; \mathcal{X}) \parallel (\mathcal{V}; \mathcal{Y}) & \mathcal{X} \cup (\mathcal{Y} \cup \mathcal{Z}) = (\mathcal{X} \cup \mathcal{Y}) \cup \mathcal{Z} \\
\mathcal{X} \cup \mathcal{X} = \mathcal{X} & \mathcal{X} \cup 0 = 0 \cup \mathcal{X} = \mathcal{X} \\
\mathcal{X} \cup \mathcal{Y} = \mathcal{Y} \cup \mathcal{X} & \mathcal{X} \parallel \mathcal{Y} = \mathcal{Y} \parallel \mathcal{X} \\
\mathcal{X} \parallel 1 = 1 \parallel \mathcal{X} = \mathcal{X} & \mathcal{X}; 1 = 1; \mathcal{X} = \mathcal{X} \\
\mathcal{X} \parallel 0 = 0 \parallel \mathcal{X} = 0 & \mathcal{X}; 0 = 0; \mathcal{X} = 0 \\
\mathcal{X} \parallel (\mathcal{Y} \cup \mathcal{Z}) = (\mathcal{X} \parallel \mathcal{Y}) \cup (\mathcal{X} \parallel \mathcal{Z}) & \mathcal{X}; (\mathcal{Y} \cup \mathcal{Z}) = (\mathcal{X}; \mathcal{Y}) \cup (\mathcal{X}; \mathcal{Z}) \\
(\mathcal{X} \cup \mathcal{Y}) \parallel \mathcal{Z} = (\mathcal{X} \parallel \mathcal{Z}) \cup (\mathcal{Y} \parallel \mathcal{Z}) & (\mathcal{X} \cup \mathcal{Y}); \mathcal{Z} = (\mathcal{X}; \mathcal{Z}) \cup (\mathcal{Y}; \mathcal{Z}) \\
\mathcal{X} \parallel (\mathcal{Y} \parallel \mathcal{Z}) = (\mathcal{X} \parallel \mathcal{Y}) \parallel \mathcal{Z} & \mathcal{X}; (\mathcal{Y}; \mathcal{Z}) = (\mathcal{X}; \mathcal{Y}); \mathcal{Z} \\
\mathcal{P}^{\parallel} = \mu\mathcal{X}.1 \cup (\mathcal{P} \parallel \mathcal{X}) & \mathcal{P}^{;} = \mu\mathcal{X}.1 \cup (\mathcal{P}; \mathcal{X}).
\end{array}$$

Proof. The details are in the accompanying technical report of this paper [21].

By Theorem 1, it makes sense to call 1 in structure $\mathfrak{S}$ the **$\bowtie$-identity program** where $\bowtie$ is a placeholder for either ; or $\parallel$. In the sequel, we call the binary relation $\subseteq$ on $\mathbb{P}$ the **program refinement relation**.

3 Least fixed point reduction

This section is about the least fixed point operators $(-)^{;}$ and $(-)^{\parallel}$. Henceforth, we shall denote these by $(-)^{\bowtie}$. We show that under a certain finiteness condition (Definition 7) the program refinement problem $\mathcal{X}^{\bowtie} \subseteq \mathcal{Y}^{\bowtie}$ can be reduced to a bounded number of program refinement problems without least fixed points (Theorem 2). To prove this, we start by inductively defining the notion of iteratively composing a program with itself under $\bowtie$.

Definition 6 (*n*-iterated-$\bowtie$-program-composition). *Let $\mathbb{N}_0 \triangleq \mathbb{N} \cup \{0\}$ be the set of **non-negative integers**. For all programs $\mathcal{P}$ in $\mathbb{P}$ and non-negative integers n in $\mathbb{N}_0$, $\mathcal{P}^{0 \cdot \bowtie} \triangleq 1 = \{\bot\}$ is the $\bowtie$-identity program and $\mathcal{P}^{(n+1) \cdot \bowtie} \triangleq \mathcal{P} \bowtie \mathcal{P}^{n \cdot \bowtie}$.*

Clearly $(-)^{\bowtie}$ is the limit of its approximations in the following sense:

Proposition 2. *For every program $\mathcal{P}$ in $\mathbb{P}$, $\mathcal{P}^{\bowtie} = \bigcup_{n \geq 0} \mathcal{P}^{n \cdot \bowtie}$.*

Definition 7 (Elementary program). *A program $\mathcal{P}$ in $\mathbb{P}$ is called **elementary** if $\mathcal{P}$ is the downward-closed set with respect to $\sqsubseteq$ of some finite and nonempty set $\mathcal{Q}$ of finite partial strings, i.e. $\mathcal{P} = \downarrow_{\sqsubseteq} \mathcal{Q}$. The set of elementary programs is denoted by $\mathbb{P}_{\ell}$.*

An elementary program therefore could be seen as a machine-representable program generated from a finite and nonempty set of finite partial strings. This finiteness restriction makes the notion of elementary programs a suitable candidate for the study of decision procedures. To make this precise, we define the following unary partial string operator:

Definition 8 (n-repeated-$\bowtie$ partial string operator). *For every non-negative integer n in $\mathbb{N}_0$, $x^{0\cdot\bowtie} \triangleq \bot$ is the empty partial string and $x^{(n+1)\cdot\bowtie} \triangleq x \bowtie x^{n\cdot\bowtie}$.*

Intuitively, $p^{n\cdot\bowtie}$ is a partial string that consists of n copies of a partial string p, each combined by the partial string operator $\bowtie$. This is formalized as follows:

Proposition 3. *Let $n \in \mathbb{N}_0$ be a non-negative integer. Define $[0] \triangleq \emptyset$ and $[n+1] \triangleq \{1, \ldots, n+1\}$. For every partial string x, $x^{n\cdot\bowtie}$ is isomorphic to $y = \langle E_y, \alpha_y, \preceq_y \rangle$ where $E_y \triangleq E_x \times [n]$ such that, for all $e, e' \in E_x$ and $i, i' \in [n]$, the following holds:*

- *if '$\bowtie$' is '$\|$', then $\langle e,\ i\rangle \preceq_y \langle e',\ i'\rangle$ exactly if $i = i'$ and $e \preceq_x e'$,*
- *if '$\bowtie$' is '$;$', then $\langle e,\ i\rangle \preceq_y \langle e',\ i'\rangle$ exactly if $i < i'$ or ($i = i'$ and $e \preceq_x e'$),*
- *$\alpha_y(\langle e,\ i\rangle) = \alpha_x(e)$.*

Definition 9 (Partial string size). *The **size** of a finite partial string p, denoted by $|p|$, is the cardinality of its event set E_p.*

For example, the partial string in Fig. 1 has size four. It is obvious that the size of finite partial strings is non-decreasing under the n-repeated-$\bowtie$ partial string operator from Definition 8 whenever $0 < n$. This simple fact is important for the next step towards our least fixed point reduction result in Theorem 2:

Proposition 4 (Elementary least fixed point pre-reduction). *For all elementary programs $\mathcal{X}$ and $\mathcal{Y}$ in $\mathbb{P}_\ell$, if the $\bowtie$-identity program 1 is not in $\mathcal{Y}$ and $\mathcal{X} \subseteq \mathcal{Y}^{\bowtie}$, then $\mathcal{X} \subseteq \bigcup_{n \geq k \geq 0} \mathcal{Y}^{k\cdot\bowtie}$ where $n = \left\lfloor \frac{\ell_\mathcal{X}}{\ell_\mathcal{Y}} \right\rfloor$ such that $\ell_\mathcal{X} \triangleq \mathsf{max}\,\{|x| \mid x \in \mathcal{X}\}$ and $\ell_\mathcal{Y} \triangleq \mathsf{min}\,\{|y| \mid y \in \mathcal{Y}\}$ is the size of the largest and smallest partial strings in $\mathcal{X}$ and $\mathcal{Y}$, respectively.*

Proof. Assume $\mathcal{X} \subseteq \mathcal{Y}^{\bowtie}$. Let $x \in \mathsf{P}_f$ be a finite partial string. We can assume $x \in \mathcal{X}$ because $\mathcal{X} \neq 0$. By assumption, $x \in \mathcal{Y}^{\bowtie}$. By Proposition 2, there exists $k \in \mathbb{N}_0$ such that $x \in \mathcal{Y}^{k\cdot\bowtie}$. Fix k to be the smallest such non-negative integer. Show $k \leq \left\lfloor \frac{\ell_\mathcal{X}}{\ell_\mathcal{Y}} \right\rfloor$ (the fraction is well-defined because $\mathcal{X}$ and $\mathcal{Y}$ are nonempty and $1 \notin \mathcal{Y}$). By downward closure and definition of $\sqsubseteq$ in terms of a one-to-one correspondence, it suffices to consider that x is one of a (not necessarily unique) longest partial strings in $\mathcal{X}$, i.e. $|x'| \leq |x|$ for all $x' \in \mathcal{X}$; equivalently, $|x| = \ell_\mathcal{X}$. If $|x| = 0$, set $k = 0$, satisfying $1 = \mathcal{X} \subseteq \mathcal{Y}^{k\cdot\bowtie} = 1$ and $k \leq n = 0$ as required. Otherwise, since the size of partial strings in a program can never decrease under the k-iterated program composition operator $\bowtie$ when $0 < k$, it suffices to consider the case $x \sqsubseteq y^{k\cdot\bowtie}$ for some shortest partial string y in $\mathcal{Y}$. Since $E_{y^{k\cdot\bowtie}}$ is the Cartesian product of E_y and $[k]$, it follows $|x| = k \cdot |y|$. Since $|x| \leq \ell_\mathcal{X}$ and $\ell_\mathcal{Y} \leq |y|$, $k \leq \left\lfloor \frac{\ell_\mathcal{X}}{\ell_\mathcal{Y}} \right\rfloor$. By definition $n = \left\lfloor \frac{\ell_\mathcal{X}}{\ell_\mathcal{Y}} \right\rfloor$, proving $x \in \bigcup_{n \geq k \geq 0} \mathcal{Y}^{k\cdot\bowtie}$. $\square$

Equivalently, if there exists a partial string x in $\mathcal{X}$ such that $x \notin \mathcal{Y}^{k\cdot\bowtie}$ for all non-negative integers k between zero and $\left\lfloor \frac{\ell_{\mathcal{X}}}{\ell_{\mathcal{Y}}} \right\rfloor$, then $\mathcal{X} \not\subseteq \mathcal{Y}^{\bowtie}$. Since we are interested in decision procedures for program refinement checking, we need to show that the converse of Proposition 4 also holds. Towards this end, we prove the following left $(-)^{\bowtie}$ elimination rule:

Proposition 5. *For every program $\mathcal{X}$ and $\mathcal{Y}$ in $\mathbb{P}$, $\mathcal{X}^{\bowtie} \subseteq \mathcal{Y}^{\bowtie}$ exactly if $\mathcal{X} \subseteq \mathcal{Y}^{\bowtie}$.*

Proof. Assume $\mathcal{X}^{\bowtie} \subseteq \mathcal{Y}^{\bowtie}$. By Proposition 2, $\mathcal{X} \subseteq \mathcal{X}^{\bowtie}$. By transitivity of $\subseteq$ in $\mathbb{P}$, $\mathcal{X} \subseteq \mathcal{Y}^{\bowtie}$. Conversely, assume $\mathcal{X} \subseteq \mathcal{Y}^{\bowtie}$. Let $i, j \in \mathbb{N}_0$. By induction on i, $\mathcal{X}^{i\cdot\bowtie} \bowtie \mathcal{X}^{j\cdot\bowtie} = \mathcal{X}^{(i+j)\cdot\bowtie}$. Thus, by Proposition 2 and distributivity of $\bowtie$ over least upper bounds in $\mathbb{P}$, $\mathcal{X}^{\bowtie} \bowtie \mathcal{X}^{\bowtie} = \mathcal{X}^{\bowtie}$, i.e. $(-)^{\bowtie}$ is idempotent. This, in turn, implies that $(-)^{\bowtie}$ is a closure operator. Therefore, by monotonicity, $\mathcal{X}^{\bowtie} \subseteq (\mathcal{Y}^{\bowtie})^{\bowtie} = \mathcal{Y}^{\bowtie}$, proving that $\mathcal{X}^{\bowtie} \subseteq \mathcal{Y}^{\bowtie}$ is equivalent to $\mathcal{X} \subseteq \mathcal{Y}^{\bowtie}$. □

Theorem 2 (Elementary least fixed point reduction). *For all elementary programs $\mathcal{X}$ and $\mathcal{Y}$ in $\mathbb{P}_\ell$, if the $\bowtie$-identity program 1 is not in $\mathcal{Y}$, then $\mathcal{X}^{\bowtie} \subseteq \mathcal{Y}^{\bowtie}$ is equivalent to $\mathcal{X} \subseteq \bigcup_{n \geq k \geq 0} \mathcal{Y}^{k\cdot\bowtie}$ where $n = \left\lfloor \frac{\ell_{\mathcal{X}}}{\ell_{\mathcal{Y}}} \right\rfloor$ such that $\ell_{\mathcal{X}} \triangleq \max\{|x| \mid x \in \mathcal{X}\}$ and $\ell_{\mathcal{Y}} \triangleq \min\{|y| \mid y \in \mathcal{Y}\}$ is the size of the largest and smallest partial strings in $\mathcal{X}$ and $\mathcal{Y}$, respectively.*

Proof. By Proposition 5, it remains to show that $\mathcal{X} \subseteq \mathcal{Y}^{\bowtie}$ is equivalent to $\mathcal{X} \subseteq \bigcup_{n \geq k \geq 0} \mathcal{Y}^{k\cdot\bowtie}$ where $n = \left\lfloor \frac{\ell_{\mathcal{X}}}{\ell_{\mathcal{Y}}} \right\rfloor$. The forward and backward implication follow from Proposition 4 and 2, respectively. □

From Theorem 2 follows immediately that $\mathcal{X}^{\bowtie} \subseteq \mathcal{Y}^{\bowtie}$ is decidable for all elementary programs $\mathcal{X}$ and $\mathcal{Y}$ in $\mathbb{P}_\ell$ because there exists an algorithm that could iteratively make $O\left(|\mathcal{X}| \times |\mathcal{Y}|^n\right)$ calls to another decision procedure to check whether $x \sqsubseteq y$ for all $x \in \mathcal{X}$ and $y \in \mathcal{Y}^{k\cdot\bowtie}$ where $n \geq k \geq 0$. However, by Proposition 1, each iteration in such an algorithm would have to solve an NP-complete subproblem. But this high complexity is expected since the PLC problem is Π_2^p-complete [12].

Corollary 1. *For all elementary programs $\mathcal{X}$ and $\mathcal{Y}$ in $\mathbb{P}$, if $|x| = |y|$ for all $x \in \mathcal{X}$ and $y \in \mathcal{Y}$, then $\mathcal{X}^{\bowtie} \subseteq \mathcal{Y}^{\bowtie}$ is equivalent to $\mathcal{X} \subseteq \mathcal{Y}$.*

We next move on to enriching our model of computation to accommodate a certain kind of relaxed sequential consistency.

4 Relaxed sequential consistency

For efficiency reasons, all modern computer architectures implement some form of weak memory model rather than sequential consistency [23]. A defining characteristic of weak memory architectures is that they violate interleaving semantics unless specific instructions are used to restore sequential consistency.

Thread $\mathtt{T}_1$		Thread $\mathtt{T}_2$
$r_0 := [b]_{\mathsf{acquire}}$	$\Vert$	$[a]_{\mathsf{none}} := 1$
$r_1 := [a]_{\mathsf{none}}$	$\Vert$	$[b]_{\mathsf{release}} := 1$

Fig. 3. A concurrent system $\mathtt{T}_1 \parallel \mathtt{T}_2$ consisting of two threads. The memory accesses on memory locations b are synchronized, whereas those on a are not.

This section fixes a particular interpretation of weak memory and studies the mathematical properties of the resulting partial order semantics. For this, we separate memory accesses into synchronizing and non-synchronizing ones, akin to [24]. A synchronized store is called a *release*, whereas a synchronized load is called an *acquire*. The intuition behind release/acquire is that prior writes made to other memory locations by the thread executing the release become visible in the thread that performs the corresponding acquire. Crucially, the particular form of release/acquire semantics that we formalize here is shown to be equivalent to the conjunction of three weak memory axioms (Theorem 3), namely 'write coherence', 'from-read' and 'global read-from' [17]. Subsequently, we look at one important ramification of this equivalence on *bounded model checking* (BMC) techniques for finding concurrency-related bugs (Theorem 4).

We start by defining the alphabet that we use for identifying events that denote synchronizing and non-synchronizing memory accesses.

Definition 10 (Memory access alphabet). *Define* $\langle LOAD \rangle \triangleq \{\mathsf{none}, \mathsf{acquire}\}$, $\langle STORE \rangle \triangleq \{\mathsf{none}, \mathsf{release}\}$ *and* $\langle BIT \rangle \triangleq \{0, 1\}$*. Let* $\langle ADDRESS \rangle$ *and* $\langle REG \rangle$ *be disjoint sets of* ***memory locations*** *and* ***registers****, respectively. Let* $load\_tag \in \langle LOAD \rangle$ *and* $store\_tag \in \langle STORE \rangle$*. Define the set of* ***load*** *and* ***store*** *labels, respectively:*

$$\Gamma_{\mathsf{load},\, load\_tag} \triangleq \{load\_tag\} \times \langle REG \rangle \times \langle ADDRESS \rangle$$

$$\Gamma_{\mathsf{store},\, store\_tag} \triangleq \{store\_tag\} \times \langle ADDRESS \rangle \times \langle BIT \rangle$$

Let $\Gamma \triangleq \Gamma_{\mathsf{load,none}} \cup \Gamma_{\mathsf{load,acquire}} \cup \Gamma_{\mathsf{store,none}} \cup \Gamma_{\mathsf{store,release}}$ *be the* ***memory access alphabet****. Given* $r \in \langle REG \rangle$, $a \in \langle ADDRESS \rangle$ *and* $b \in \langle BIT \rangle$*, we write '*$r := [a]_{load\_tag}$*' for the label* $\langle load\_tag, r, a \rangle$ *in* $\Gamma_{\mathsf{load},\, load\_tag}$*; similarly, '*$[a]_{store\_tag} := b$*' is shorthand for the label* $\langle store\_tag, a, b \rangle$ *in* $\Gamma_{\mathsf{store},\, store\_tag}$*.*

Let x *be a partial string and* e *be an event in* E_x*. Then* e *is called a* ***load*** *or* ***store*** *if its label,* $\alpha_x(e)$*, is in* $\Gamma_{\mathsf{load},\, load\_tag}$ *or* $\Gamma_{\mathsf{store},\, store\_tag}$*, respectively. A load or store event* e *is a* ***non-synchronizing memory access*** *if* $\alpha_x(e) \in \Gamma_{\mathsf{none}} \triangleq \Gamma_{\mathsf{load,none}} \cup \Gamma_{\mathsf{store,none}}$*; otherwise, it is a* ***synchronizing memory access****. Let* $a \in \langle ADDRESS \rangle$ *be a memory location. An* ***acquire on*** a *is an event* e *such that* $\alpha_x(e) =$ *'*$r := [a]_{\mathsf{acquire}}$*' for some* $r \in \langle REG \rangle$*. Similarly, a* ***release on*** a *is an event* e *labelled by '*$[a]_{\mathsf{release}} := b$*' for some* $b \in \langle BIT \rangle$*. A* ***release*** *and* ***acquire*** *is a release and acquire on some memory location, respectively.*

Example 5. Fig. 3 shows the syntax of a program that consists of two threads $\mathtt{T}_1$ and $\mathtt{T}_2$. This concurrent system can be directly modelled by the partial string shown in Fig. 1 where memory location b is accessed through acquire and release, whereas memory location a is accessed through non-synchronizing loads and stores (shortly, we shall see that this leads to a data race).

Given Definition 10, we are now ready to refine our earlier conservative over-approximation of the happens-before relations (Definition 4) to get a particular form of release/acquire semantics. For this, we restrict the downward closure of programs $\mathcal{X}$ in $\mathbb{P}$, in the sense of Definition 4, by requiring all partial strings in $\mathcal{X}$ to satisfy the following partial ordering constraints:

Definition 11 (SC-relaxed program). *A program $\mathcal{X}$ is called **SC-relaxed** if, for all $a \in \langle ADDRESS \rangle$ and partial string x in $\mathcal{X}$, the set of release events on a is totally ordered by $\preceq_x$ and, for every acquire $l \in E_x$ and release $s \in E_x$ on a, $l \preceq_x s$ or $s \preceq_x l$.*

Henceforth, we denote loads and stores by l, l' and s, s', respectively. If s and s' are release events that modify the same memory location, either s happens-before s', or vice versa. If l is an acquire and s is a release on the same memory location, either l happens-before s or s happens-before l. Importantly, however, two acquire events l and l' on the same memory location may still happen concurrently in the sense that neither l happens-before l' nor l' happens-before l, in the same way non-synchronizing memory accesses are generally unordered.

Example 6. Example 4 and 5 illustrate the SC-relaxed semantics of the concurrent system in Fig. 3. In particular, the N-shaped partial string in Fig. 2 corresponds to a data race in $\mathtt{T}_1 \parallel \mathtt{T}_2$ because the non-synchronizing memory accesses on memory location a happen concurrently. To see this, it may help to consider the interleaving $r_0 := [b]_{\mathsf{acquire}}; [a]_{\mathsf{none}} := 1; r_1 := [a]_{\mathsf{none}}; [b]_{\mathsf{release}} := 1$ where both memory accesses on location a are unordered through the happens-before relation because there is no release instruction separating $[a]_{\mathsf{none}} := 1$ from $r_1 := [a]_{\mathsf{none}}$. One way of fixing this data race is by changing thread $\mathtt{T}_1$ to **if** $[b]_{\mathsf{acquire}} = 1$ **then** $r_1 := [a]_{\mathsf{none}}$. Since CKA supports non-deterministic choice with the $\cup$ binary operator (recall Theorem 1), it would not be difficult to give semantics to such conditional checks, particularly if we introduce 'assume' labels into the alphabet in Definition 10.

We ultimately want to show that the conjunction of three existing weak memory axioms as studied in [17] fully characterizes our particular interpretation of relaxed sequential consistency, thereby paving the way for Theorem 4. For this, we recall the following memory axioms which can be thought of as relations on loads and stores on the same memory location:

Definition 12 (Memory axioms). *Let x be a partial string in P_f. The **read-from** function, denoted by $\mathsf{rf}\colon E_x \to E_x$, is defined to map every load to a store on the same memory location. A load l **synchronizes-with** a store s if $\mathsf{rf}(l) = s$ implies $s \preceq_x l$. **Write-coherence** means that all stores s, s' on the same memory location are totally ordered by $\preceq_x$. The **from-read axiom** holds whenever, for all loads l and stores s, s' on the same memory location, if $\mathsf{rf}(l) = s$ and $s \prec_x s'$, then $l \preceq_x s'$.*

By definition, the read-from function is total on all loads. The synchronizes-with axiom says that if a load reads-from a store (necessarily on the same memory location), then the store happens-before the load. This is also known as the global read-from axiom [17]. Write-coherence, in turn, ensures that all stores on the same memory location are totally ordered. This corresponds to the fact that "all writes to the same location are serialized in some order and are performed in that order with respect to any processor" [24]. Note that this is different from the modification order ('mo') on atomics in C++14 [25] because 'mo' is generally not a subset of the happens-before relation. The from-read axiom [17] requires that, for all loads l and two different stores s, s' on the same location, if l reads-from s and s happens-before s', then l happens-before s'.We start by deriving from these three memory axioms the notion of SC-relaxed programs.

Proposition 6 (SC-relaxed consistency). *For all $\mathcal{X}$ in $\mathbb{P}$, if, for each partial string x in $\mathcal{X}$, the synchronizes-with, write-coherence and from-read axioms hold on all release and acquire events in E_x on the same memory location, then $\mathcal{X}$ is an SC-relaxed program.*

Proof. Let $a \in \langle ADDRESS \rangle$ be a memory location, l be an acquire on a and s' be a release on a. By write-coherence on release/acquire events, it remains to show $l \preceq_x s'$ or $s' \preceq_x l$. Since the read-from function is total, $\mathsf{rf}(l) = s$ for some release s on a. By the synchronizes-with axiom, $s \preceq_x l$. We therefore assume $s \neq s'$. By write-coherence, $s \prec_x s'$ or $s' \prec_x s$. The former implies $l \preceq_x s'$ by the from-read axiom, whereas the latter implies $s' \preceq_x l$ by transitivity. This proves, by case analysis, that $\mathcal{X}$ is an SC-relaxed program. □

We need to prove some form of converse of the previous implication in order to characterize SC-relaxed semantics in terms of the three aforementioned weak memory axioms. For this purpose, we define the following:

Definition 13 (Read consistency). *Let $a \in \langle ADDRESS \rangle$ be a memory location and x be a finite partial string in P_f. For all loads $l \in E_x$ on a, define the following set of store events: $\mathcal{H}_x(l) \triangleq \{s \in E_x \mid s \preceq_x l \text{ and } s \text{ is a store on } a\}$. The read-from function rf is said to satisfy **weak read consistency** whenever, for all loads $l \in E_x$ and stores $s \in E_x$ on memory location a, the least upper bound $\bigvee \mathcal{H}_x(l)$ exists, and $\mathsf{rf}(l) = s$ implies $\bigvee \mathcal{H}_x(l) \preceq_x s$; **strong read consistency** implies $\mathsf{rf}(l) = s = \bigvee \mathcal{H}_x(l)$.*

By the next proposition, a natural sufficient condition for the existence of the least upper bound $\bigvee \mathcal{H}_x(l)$ is the finiteness of the partial strings in P_f and the total ordering of all stores on the same memory location from which the load l reads, i.e. write coherence. This could be generalized to well-ordered sets.

Proposition 7 (Weak read consistency existence). *For all partial strings x in P_f, write coherence on memory location a implies that $\bigvee \mathcal{H}_x(l)$ exists for all loads l on a.*

We remark that $\bigvee \mathcal{H}_x(l) = \bot$ if $\mathcal{H}_x(l) = \emptyset$; alternatively, to avoid that $\mathcal{H}_x(l)$ is empty, we could require that programs are always constructed such that their partial strings have minimal store events that initialize all memory locations.

Proposition 8 (Weak read consistency equivalence). *Write coherence implies that weak read consistency is equivalent to the following: for all loads l and stores s, s' on memory location $a \in \langle ADDRESS \rangle$, if $\mathsf{rf}(l) = s$ and $s' \preceq_x l$, then $s' \preceq_x s$.*

Proof. By write coherence, $\bigvee \mathcal{H}_x(l)$ exists, and $s' \preceq_x \bigvee \mathcal{H}_x(l)$ because $s' \in \mathcal{H}_x(l)$ by assumption $s' \preceq_x l$ and Definition 13. By assumption of weak read consistency, $\bigvee \mathcal{H}_x(l) \preceq_x s$. From transitivity follows $s' \preceq_x s$.

Conversely, assume $\mathsf{rf}(l) = s$. Let s' be a store on a such that $s' \in \mathcal{H}_x(l)$. Thus, by hypothesis, $s' \preceq_x s$. Since s' is arbitrary, s is an upper bound. Since the least upper bound is well-defined by write coherence, $\bigvee \mathcal{H}_x(l) \preceq_x s$. □

Weak read consistency therefore says that if a load l reads from a store s and another store s' on the same memory location happens before l, then s' happens before s. This implies the next proposition.

Proposition 9 (From-read equivalence). *For all SC-relaxed programs in $\mathbb{P}$, weak read consistency with respect to release/acquire events is equivalent to the from-read axiom with respect to release/acquire events.*

We can characterize strong read consistency as follows:

Proposition 10 (Strong read consistency equivalence). *Strong read consistency is equivalent to weak read consistency and the synchronizes-with axiom.*

Proof. Let x be a partial string in P_f. Let l be a load and s be a store on the same memory location. The forward implication is immediate from $\bigvee \mathcal{H}_x(l) \preceq_x l$.

Conversely, assume $\mathsf{rf}(l) = s$. By synchronizes-with, $s \preceq_x l$, whence $s \in \mathcal{H}_x(l)$. By definition of least upper bound, $s \preceq_x \bigvee \mathcal{H}_x(l)$. Since $s \succeq_x \bigvee \mathcal{H}_x(l)$, by hypothesis, and $\preceq_x$ is antisymmetric, we conclude $s = \bigvee \mathcal{H}_x(l)$. □

Theorem 3 (SC-relaxed equivalence). *For every program $\mathcal{X}$ in $\mathbb{P}$, $\mathcal{X}$ is SC-relaxed where, for all partial strings x in $\mathcal{X}$ and acquire events l in E_x, $\mathsf{rf}(l) = \bigvee \mathcal{H}_x(l)$, if and only if the synchronizes-with, write-coherence and from-read axioms hold for all x in $\mathcal{X}$ with respect to all release/acquire events in E_x on the same memory location.*

Proof. Assume $\mathcal{X}$ is an SC-relaxed program according to Definition 11. Let x be a partial string in $\mathcal{X}$ and l be an acquire in the set of events E_x. By Proposition 7, $\bigvee \mathcal{H}_x(l)$ exists. Assume $\mathsf{rf}(l) = \bigvee \mathcal{H}_x(l)$. Since l is arbitrary, this is equivalent to assuming strong read consistency. Since release events are totally ordered in $\preceq_x$, by assumption, it remains to show that the synchronizes-with and from-read axioms hold. This follows from Proposition 10 and 9, respectively.

Conversely, assume the three weak memory axioms hold on x with respect to all release/acquire events in E_x on the same memory location. By Proposition 6, $\mathcal{X}$ is an SC-relaxed program. Therefore, by Proposition 9 and 10, $\mathsf{rf}(l) = \bigvee \mathcal{H}_x(l)$, proving the equivalence. □

While the state-of-the-art weak memory encoding is cubic in size [18], the previous theorem has as immediate consequence that there exists an asymptotically smaller weak memory encoding with only a quadratic number of partial order constraints.

Theorem 4 (Quadratic-size weak memory encoding). *There exists a quantifier-free first-order logic formula that has a quadratic number of partial order constraints and is equisatisfiable to the cubic-size encoding given in [18].*

Proof. Instead of instantiating the three universally quantified events in the from-read axiom, symbolically encode the least upper bound of weak read consistency. This can be accomplished with a new symbolic variable for every acquire event. It is easy to see that this reduces the cubic number of partial order constraints to a quadratic number. □

In short, the asymptotic reduction in the number of partial order constraints is due to a new symbolic encoding for how values are being overwritten in memory: the current cubic-size formula [18] encodes the from-read axiom (Definition 12), whereas the proposed quadratic-size formula encodes a certain least upper bound (Definition 13). We reemphasize that this formulation is in terms of release/acquire events rather than machine-specific accesses as in [18]. The construction of the quadratic-size encoding, therefore, is generally only applicable if we can translate the machine-specific reads and writes in a shared memory program to acquire and release events, respectively. This may require the program to be data race free, as illustrated in Example 6.

Furthermore, as mentioned in the introduction of this section, the primary application of Theorem 4 is in the context of BMC. Recall that BMC assumes that all loops in the shared memory program under scrutiny have been unrolled (the same restriction as in [18]). This makes it possible to symbolically encode branch conditions, thereby alleviating the need to explicitly enumerate each finite partial string in an elementary program.

5 Concluding remarks

This paper has studied a partial order model of computation that satisfies the axioms of a unifying algebraic concurrency semantics by Hoare et al. By further restricting the partial string semantics, we obtained a relaxed sequential consistency semantics which was shown to be equivalent to the conjunction of three weak memory axioms by Alglave et al. This allowed us to prove the existence of an equisatisfiable but asymptotically smaller weak memory encoding that has only a quadratic number of partial order constraints compared to the state-of-the-art cubic-size encoding. In upcoming work, we will experimentally compare both encodings in the context of bounded model checking using SMT solvers. As future theoretical work, it would be interesting to study the relationship between categorical models of partial string theory and event structures.

Acknowledgements. We would like to thank Tony Hoare and Stephan van Staden for their valuable comments on an early draft of this paper, and we thank Jade Alglave, César Rodríguez, Michael Tautschnig, Peter Schrammel, Marcelo Sousa, Björn Wachter and John Wickerson for invaluable discussions.

References

1. Hoare, C.A., Möller, B., Struth, G., Wehrman, I.: Concurrent Kleene algebra and its foundations. J. Log. Algebr. Program. **80**(6) (2011) 266–296

2. Hoare, T., van Staden, S.: The laws of programming unify process calculi. Sci. Comput. Program. **85** (2014) 102–114
3. Hoare, T., van Staden, S.: In praise of algebra. Formal Aspects of Computing **24**(4-6) (July 2012) 423–431
4. Hoare, T., van Staden, S., Möller, B., Struth, G., Villard, J., Zhu, H., W. O'Hearn, P.: Developments in Concurrent Kleene Algebra. RAMiCS '14 (2014) 1–18
5. Pratt, V.: Modeling concurrency with partial orders. Int. J. Parallel Program. **15**(1) (February 1986) 33–71
6. Gischer, J.L.: The equational theory of pomsets. Theor. Comput. Sci. **61**(2-3) (November 1988) 199–224
7. Petri, C.A.: Communication with automata. PhD thesis, Universität Hamburg (1966)
8. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7) (July 1978) 558–565
9. Grabowski, J.: On partial languages. Fundam. Inform. **4**(2) (1981) 427–498
10. Nielsen, M., Plotkin, G.D., Winskel, G.: Petri nets, event structures and domains, part I. Theor. Comput. Sci. **13**(1) (1981) 85 – 108
11. Bloom, B., Kwiatkowska, M.Z.: Trade-offs in true concurrency: Pomsets and Mazurkiewicz traces. MFPS '91, Springer (1992) 350–375
12. Feigenbaum, J., Kahn, J., Lund, C.: Complexity results for POMSET languages. SIAM J. Discret. Math. **6**(3) (1993) 432–442
13. Laurence, M.R., Struth, G.: Completeness theorems for Bi-Kleene algebras and series-parallel rational pomset languages. RAMiCS '14 (2014) 65–82
14. Sewell, P., Sarkar, S., Owens, S., Nardelli, F.Z., Myreen, M.O.: x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. Commun. ACM **53**(7) (July 2010) 89–97
15. Ševčík, J., Vafeiadis, V., Zappa Nardelli, F., Jagannathan, S., Sewell, P.: Relaxed-memory concurrency and verified compilation. SIGPLAN Not. **46**(1) (January 2011) 43–54
16. Batty, M., Owens, S., Sarkar, S., Sewell, P., Weber, T.: Mathematizing C++ concurrency. SIGPLAN Not. **46**(1) (January 2011) 55–66
17. Alglave, J., Maranget, L., Sarkar, S., Sewell, P.: Fences in weak memory models (extended version). FMSD **40**(2) (2012) 170–205
18. Alglave, J., Kroening, D., Tautschnig, M.: Partial orders for efficient bounded model checking of concurrent software. CAV'13, Springer (2013) 141–157
19. Ésik, Z.: Axiomatizing the subsumption and subword preorders on finite and infinite partial words. Theor. Comput. Sci. **273**(1-2) (February 2002) 225–248
20. Alglave, J., Maranget, L., Sarkar, S., Sewell, P.: Litmus: Running tests against hardware. TACAS'11/ETAPS'11, Springer (2011) 41–44
21. Horn, A., Alglave, J.: Concurrent Kleene algebra of partial strings. ArXiv e-prints **abs/1407.0385** (July 2014)
22. de Bakker, J.W., Warmerdam, J.H.A.: Metric pomset semantics for a concurrent language with recursion. In: Proceedings of the LITP Spring School on Theoretical Computer Science on Semantics of Systems of Concurrent Processes. (1990) 21–49
23. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Comput. **28**(9) (September 1979) 690–691
24. Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A., Hennessy, J.: Memory consistency and event ordering in scalable shared-memory multiprocessors. SIGARCH Comput. Archit. News **18**(2SI) (May 1990) 15–26
25. ISO: International Standard ISO/IEC 14882:2014(E) Programming Language C++. International Organization for Standardization (2014) Ratified, to appear soon.